CS 550 Project Report

Introduction

In this project, we touched upon the major concepts in Coq and demonstrated the running and interpretation of code in each part of the presentation. We are thankful for being given the opportunity to conduct research into such an interesting area of computing and for gaining exposure to a completely new programming paradigm. In traditional programming languages like C, Java, and Python, the computer is not capable of making inferences and deriving proofs. It only operates on the principles that we program it with. It was interesting to see how the power of computers can be extended into the realm of logical and mathematical proofs.

The common structure of our presentation was that we began with discussion of the mathematical foundations of components such as Imp, Small Step, and STLC, and demonstrated their encoding into informal grammatical notations and formal Coq implementations. We ended by showing the parsing of the relevant proofs from Software Foundations, and presented the exact mechanism of their working in order to elucidate the working of Coq on a macro level. The final presentation tried to unify these concepts to present an overarching view of the importance of Coq in language study and validation of formal mathematical models and their properties.

Imperative Programming

Imperative programming is a well-known and widely used paradigm. But we never truly understood what exactly goes on underneath the hood of the compiler. Being afforded the opportunity to study the mechanics of imperative programming has brought us a much clearer appreciation of the issues involved with it.

We started with the representation of syntax for arithmetic and Boolean operations using rules in BFN notation and in Coq. This was followed by the actual total order arithmetic function evaluation in Coq. This line of study led us into an important study of the optimization of the parse tree generated by the arithmetic grammar and our first important proof on the correctness of the optimization function **– Example 1 of IMP.**

This proof led us in the area of study to use powerful tools and tactics using Coq Automation to be able to generate complex proofs for optimizations in more real world problems. Our first proof was showed in a more condensed manner in conjunction with the study of Tacticals such as Try, Repeat, ‘;’ etc. to demonstrate the features in the proof development capabilities of Coq. The limitations in the former view of functions led us to a more expressive mode of function evaluation in imp using relations. Relations guided us through to relate the intentional and extensional model of function evaluation and both their advantages and disadvantages using Coq.

We listed and explained the relational (extensional) vs computational (intentional) definitions of functions, explained both their theoretical differences and started with their syntactic differences and further elaborated on their semantic usage in real world programming languages using Coq. We added important features to the relational view such as non-determinism, and moved onto the study of machine states by introducing variables, their scope and evaluation. Finally, we joined all the dots together and moved onto defining the syntax and semantics of Imp commands (statements).

We explained the failed attempt of evaluation as a function and then correlated our initial introduction of the relational definition of functions as a tool for constructing the formal rules of evaluation of commands in Imperative programming languages using Coq. We introduced these rules and then explained their operational semantics.

Our analysis did not stop at this introduction and we moved onto probably the most important proof in the project and in the construction of our previous grammar in Coq i.e. The deterministic behavior of the outcome of our commands encoded in Coq using partial functions (relations.) **– Example 2 of IMP.**

Small Step Semantics

We are trained to think of functions and programs as black boxes; they take some input and spit out some result. It is indeed simpler to think of them as monolithic blocks or immutable execution paths, but this misses much of the nuance that goes into their design and implementation. Viewing them as a sum of smaller components gives one a better understanding of the exact mechanics that make them useful and functional.

We introduced a big step style of semantics in the earlier paradigm where input states are mapped onto output states with no control over intermediate states in the evaluation process. This led us to study the concept of small step semantics. We explored a serious deficiency in the usage of big step semantics where an inductive definition of function rules applied on partial functions might lead to an infinite loop or a stuck program where the order and types of arguments make none of the rules of the evaluation process applicable.

Infinite looping and erroneous configuration of states are very different problems of language design. In this part of the presentation we introduced and applied the small step semantics in the partial function evaluation process to distinguish between these 2 kinds of errors before type checking could be possibly applied to the latter in language design.

We introduced the notion of a term and achieved this finer level of control of states using new rules to guide the evaluation in a left-to-right order starting from the leaves of the parse tree where the base case considers values and the inductive cases considers evaluation of left terms followed by evaluation of right terms in the expression. We expressed these semantics using informal BNF notations followed by the formal grammar written in Coq. This brought us to the deterministic behavior of rules in a single step of the Small Step evaluation process – **Example 1 in Small Step.** Next, we formally differentiated between terms and values to arrive at the halting state of the abstract machine where terms are used to represent a state of the machine and the value is a special case of the term like a special symbol to identify the halting state of the machine in the end when all the terms have been reduced to values.

We discussed the notion of strong progress in small step semantics which is a case of the generalized progress in Simply Typed Lambda Calculus where strong here meant that the term is a value or can make 1 step onto another term. After this, we moved into formally specify the reduced form as the normal form for the term where terms cannot make further progress. We also tried to make distinction between normal forms and values where values where syntactic notations and normal forms carried semantic meaning associated to them. The study of single step reduction naturally paved the way for introducing multi-step reduction also called beta-reduction in lambda calculus.

Parallelism

The possibility of parallel execution follows from the flexibility inherent in small step semantic systems. The possibilities that arise from the compiler being able to conduct out of order execution of instructions brings up the possibility of great performance gains. The problems that arise from the non-determinism of such an execution model enlightens one to the need for concurrency control mechanisms like semaphores, mutexes, scheduling etc.

We tried to enhance the Big-step version of Imp arithmetic and Boolean operations using the tools of small step. We achieved these by incorporating terms, values, and finally, the operational semantics of small step using 3 subcases for each rule with the base case for values and inductive cases for left to right term evaluation in each operation **– Example 2 in small step.** This was represented in depth using the Coq Ide. We then dealt with the issue of commands in Imp.

The final issue in Small Step was to study an application for small step semantics using its fine grained control. This application in language design was the study of Concurrent Models. We introduced rules for Concurrent evaluation of Imp programs, stated an example in Concurrent form and demonstrated the proofs for multiple values as a result of the output variable of the imperative program – **Example 3 in small step.** This exploration in the direction of Imp and Small step equipped us to deal with the fundamentals of introducing Simply Typed Lambda Calculus in Coq.

Simply Typed Lambda Calculus

The complexity that underlies lambda calculus is not immediately obvious when approaching it through programming languages with functional paradigms. Looking at it from a purely scientific direction was eye opening, and we have begun to see it as a powerful and valid mathematical and logical system in its own right, rather than an opaque and esoteric set of rules that is only useful when used within a programming language. Understanding the behavior of a given language goes beyond simply knowing the general syntax. To fully grasp how symbols of an alphabet align in order to communicate instructions to a machine, one must understand the underlying comprehension process.

One must consider how machines consume symbols and how this affects the state of the system. With deterministic finite automata, a system is described as a having a set of states (with a starting state, accepting states, intermediate states), and given alphabet that the system can process and transition functions that change the state of the system depending on the input symbol. Nondeterministic finite automata function in a similar way, but have a range of possible transition functions for a given input symbol. With both, however, the acceptance of string ultimately depends on whether or not the system ends in an accept state.

One must consider what is considered a legal arrangement of inputs. Context-free grammars and syntax trees allow for the conceptualization of these rules. With a program's grammar, a start symbol is defined as the entire top-level phrase being parsed, which can then be broken down into a final expression based on the productions. This can be visualized as a parse tree or through Backus-Naur Form notation, which define what can possibly be produced by given symbol. A common example of this process is a postal address. The main address can be broken down into a name component, a street address component and zip code component. Naturally, each of these components (non-terminals) are broken down further via productions that define what is valid for a name, street address and zip code until they reach their most basic components (terminals).

There are numerous ways in which input strings can be compared against the aforementioned grammars of a given language. A bottom-up parser matches the input against the terminals of a parse tree and reduces them to the appropriate non-terminal repeatedly until it cannot be collapsed further via a valid grammar convention. If the string can be logically reformed into the starting token, the string is considered valid. On the other hand, a top-down parser begins with the start token and reduces it via productions in order to determine if the string can be produced. Naturally, if it can be produced, the string is considered valid.

This would be sufficient if an input string was simply being evaluated for validity in a given language, but in the real world, the aforementioned strings aren't simply arrangements of symbols. They're units of data that have specific types and conventions for how they can be manipulated. Operational semantics and logical programming are used to define procedural relationships between statements. These relationships are verified via the construction of proofs concerning the execution of code within a given language. This process of proving relationships between statements is crucial because it maintains a consistent logic. Without this logic, languages would be susceptible to inconsistencies and paradoxical uses that could fatally compromise their robustness.

Typed languages also draw from the Simply-typed lambda calculus, whose rules form the basis for how units of data can be manipulated with consideration to type, scope, and relationships in addition to embodying functional abstraction. The STLC also provides the framework for higher-order functionality—that is the ability input functions into other functions and ultimately derive logical information. This is achieved through Its two most powerful features, substitution and reduction. These features describe, with consideration to scope and type, how a statement can be altered and simplified while maintaining its logical integrity, which ultimately describes the functionality of all typed programming languages—i.e. a symbol is inputted into a function, checked for validity and used to evaluate the body of the function if appropriate. The application of all of the above features of programming language evaluation is crucial for maintaining a consistent logic and assuring functional performance.

STLC in Coq

The sheer scale and complexity of the problem of imposing an order and hierarchy of types and properties to raw data cannot be fully appreciated without a detailed look at the exact mechanism that is used to do it. We have observed the rules used to impose simple typing, as well as the mechanisms that are used to extend these simple types to hold complex data, as well as introduce data hierarchy.

The key ideas we explored in STLC were to use the tools to the syntax, small step semantics and typing in Coq to introduce STLC and study the challenges of variable binding and substitution in the STLC paradigm of Coq. We ended our presentation with a note on the expansion of STLC in Coq and important properties of the Typing contexts and its relevance to Types in Language design.

We started by introducing the concept of variables, abstractions, applications, Booleans and conditionals relevant to lambda calculus. We dealt only with bools initially with a clear distinction for arrow types to classify functions in our design. This led into some key examples in application of abstractions and their syntactic representation in Coq. We then moved onto the critical notions of free variables and substitution in STLC where we first modified values to be absolute, true or false which were relevant to STLC design at this point. The small step semantics of Coq allowed us to distinguish between values and variables without explicitly defining it in the context of STLC.

The heart of STLC is substitution. We represented the informal rules as before and then considered their formal representation in Coq with a point on the similarity in representation of these formal and informal rules, which highlights the expressive power of Coq. While discussing reduction, small step semantics allow us to translate the informal rules of reduction in STLC to their formal representations in Coq. We then incorporated rules for types in STLC using Typing Contexts and explained its significance in terms of formal rules and again demonstrated the ease of translation of these rules in Coq.

These basics of STLC led us to an important argument on the properties of rules in STLC and how useful it is to design STLC in Coq for a sound proof of these properties. A clear distinction between soundness, progress and preservation was made where the properties of soundness nicely integrated into small step semantics and the emphasis of well typed terms of axioms in program termination was discussed.

We discussed the importance of the type preservation proof, the substitution lemma, the context invariance lemma and the free variables of term proof where we discussed how useful Coq is in the validation of constructs in the STLC. The final phase of the discussion on STLC led us to demonstrate how types in the STLC design of Coq allows us to easily create new types such as pairs, records and so on with an example on pairs and evaluation and typing rules of pairs and their corresponding elements (projections).

Conclusion

Coq as discussed is a Proof Assistant System with varied applications in validation of Language Design and validation of formal mathematical constructs. It has been used to prove the essential properties of the Java Card system and formalize the semantics of the data oriented flow synchronous calculus amongst its most noted applications.

The Kernel for the Coq Proof Assistant system is so small and concise yet powerful in its expressive capabilities. The reason for this system design is because Coq is a consistent system, it is used in the proof of important constructs. If the system of Coq is rendered vulnerable to software and hardware bugs, then the entire Coq system is deemed invalid and obsolete. Therefore, there is little scope for extensions to Coq; the main motivation for this project was to theoretically state and practically demonstrate via proofs some of the most important constructs of language design and their properties.